What is a hash?

In simplest terms, a hash is a fixed size integer which identifies a particular value. Please note that this is the simplest explanation.

Let us point out what a fixed hash can mean:

- Same data will have same hash value.
- Even a slight change in original data can result in a completely different hash value.
- A hash is obtained from a hash function, whose responsibility is to convert the given information to encoded hash.
- Clearly, the number of objects can be much more than the number of hash values and so, two objects may hash to the same hash value. This is called **Hash collision**. This means that **if two objects have the same hash code, they do not necessarily have the same value**.

What is Python hash function?

We can move into great detail about hashing but an important point about making a **GOOD Hash function** is worth mentioning here:

> A good hash function is the one which results in the least number of collisions, meaning, no 2 set of information should have the same hash values.

Apart from the above definition, hash value of an object should be cheap to calculate in terms of space and memory complexity.

Hash codes are most used in when comparison for dictionary keys is done. Hash code of dictionary keys is compared when dictionary lookup is done for a specific key. Comparing hash is much faster than comparing the complete key values because the set of integers that the hash function maps each dictionary key to is much smaller than the set of objects itself.

Also, note that if two numeric values can compare as equal, they will have the same hash as well, even if they belong to different data types, like 1 and 1.0.

Python hash() String

Let us start constructing simple examples and scenarios in which the `hash()`function can be very helpful. In this example, we will simply get the hash value of a String.

```
name = "Shubham"

hash1 = hash(name)
hash2 = hash(name)

print("Hash 1: %s" % hash1)
print("Hash 2: %s" % hash2)
```

We will obtain the following result when we run this script:

python hash string, python hash function

Here is an important catch. If you run the same script again, the hash changes as shown below:

python hash() example

So, the life of a hash is only for the program scope and it can change as soon as the program has ended.

Python hash with slight change in data

Here, we will see how a slight change in data can change a hash value. Will it change completely or just a little? A better way is to find out through a script!

```
name1 = "Shubham"
name2 = "Shubham!"

hash1 = hash(name1)
hash2 = hash(name2)

print("Hash 1: %s" % hash1)
print("Hash 2: %s" % hash2)
```

Let’s run this script now:

python hash int value

See how the hash changed completely when only one character changed in original data? This makes a hash value completely unpredictable!

How to define hash() function for custom objects?

Internally, `hash()`function works by overriding the `__hash__()`function. It is worth noticing that not every object is hashable (mutable collections aren't hashable). We can also define this function for our custom class. Actually, that's what we will do now. Before that, let's point out some important points:

- Hashable implementation should not be done for mutable collections as key's of collections should be immutablefor hashing.
- We don't have to define a custom `__eq__()`function implementation as it is defined for all objects.

Now, let us define an object and override the `__hash__()`function:

```
class Student:
    def __init__(self, age, name):
        self.age = age
        self.name = name

    def __eq__(self, other):
        return self.age == other.age and self.name == other.name

    def __hash__(self):
        return hash((self.age, self.name))

student = Student(23, 'Shubham')
print("The hash is: %d" % hash(student))
```

Let's run this script now:

python hash object

This program actually described how we can override both the `__eq__()`and the `__hash__()`functions. This way, we can actually define our own logic to compare any objects.

Why mutable objects cannot be Hashed?

As we already know, only immutable objects can be hashed. This restriction of not allowing a mutable object to be hashed simplify the hash table a lot. Let’s understand how.

If a mutable object is allowed to be hashed, we need to update the hash table every time the value of the objects updates. This means that we will have to move the object to a completely different bucket. This is a very costly operation to be performed.

In Python, we have two objects that uses hash tables, dictionariesand sets:

- A dictionary is a hash table and is called an associative array. In a dictionary, only keys are hashed and not the values. This is why a dictionary key should be an immutable object as well while values can be anything, even a list which is mutable.
- A set contains unique objects which are hashable. If we have non-hashable items, we cannot use set and must instead use list.

That’s all for a quick roundup on python hash() function.